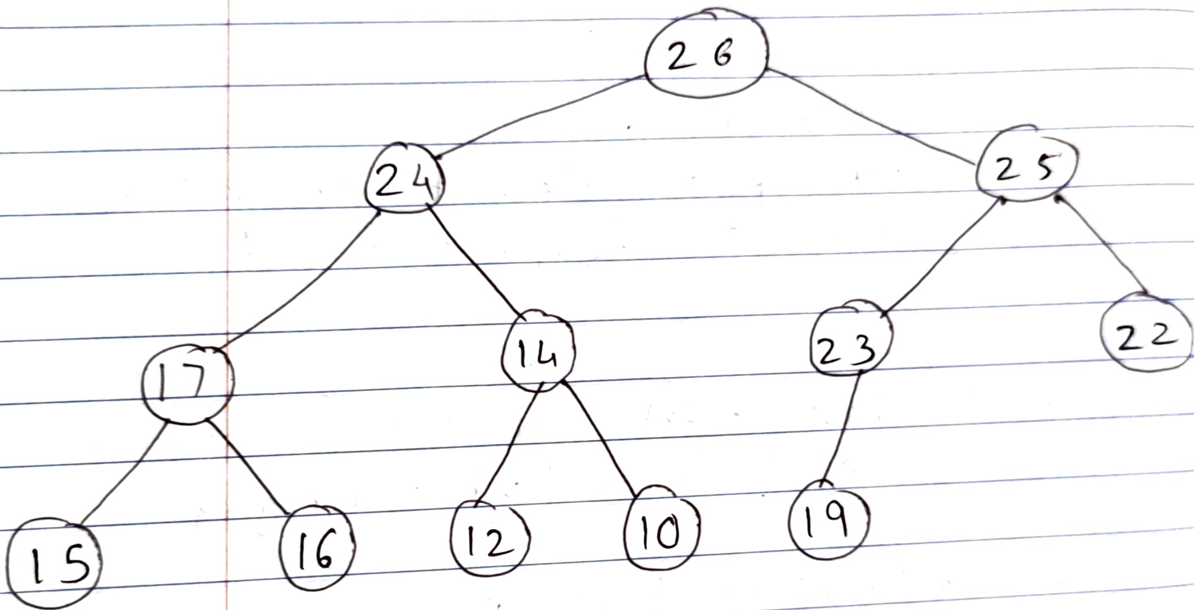
~~Howe~~ Homework 4

Q 1]

a) Final max-heap:



Array:

< 26, 24, 25, 17, 14, 23, 22, 15, 16, 12,

10, 19 >

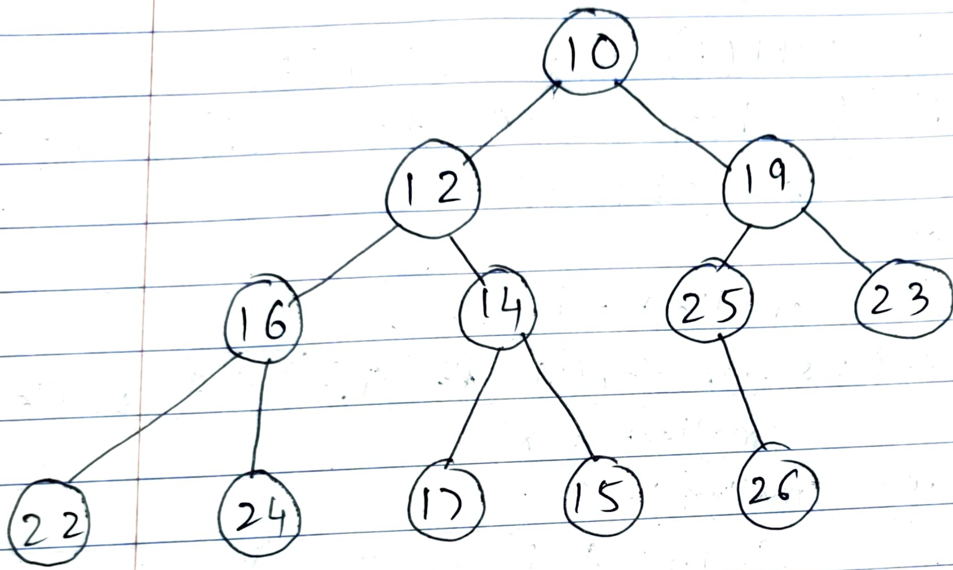
b) Min-Heapify (A, i)

```
{
    l = Left(i) ; r = Right(i)
    if ( l <= heap_size (A) && A[l] < A[i]
            smallest = l;
    else
            smallest = i;
    if ( r ≤ heap_size(A) && A[r] < A [smallest]
            smallest = r;
    if   smallest != i
            Swap ( A, i, smallest )
            Min-Heapify (A, smallest)
}
```

```
Build-Min-Heap ( A )
{
        n = length (A)
        for i ← ⌊n/2⌋ downto 1
            do  Min-Heapify (A, i, n)
}
```

c)

Final min-heap:



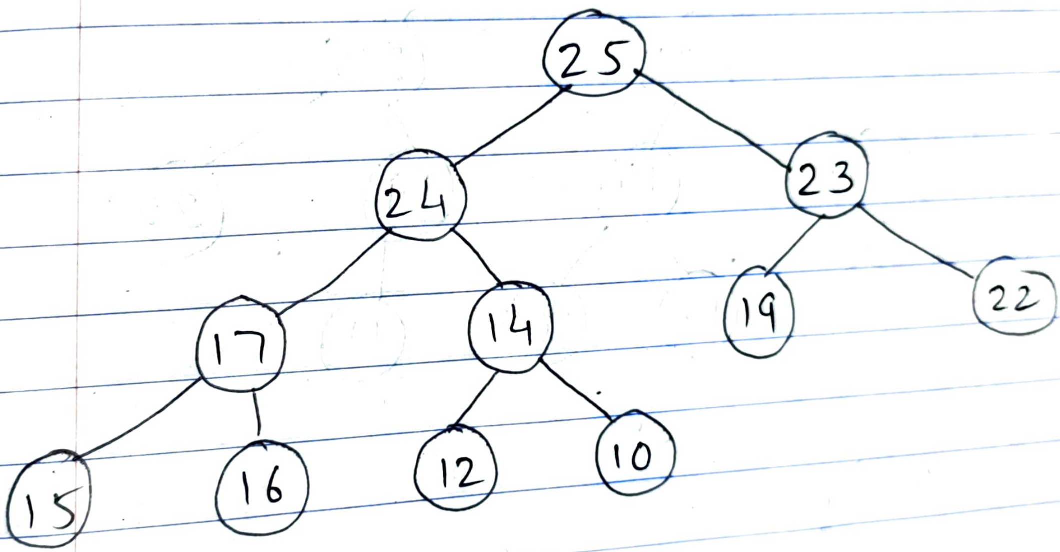
Array:

< 10, 12, 19, 16, 14, 25, 23, 22, 24,

17, 15, 26 >

d) Extract max discards the root element and the last leaf node becomes the new root. Then max-heapify is applied ~~to the root node~~ to heapify the tree.

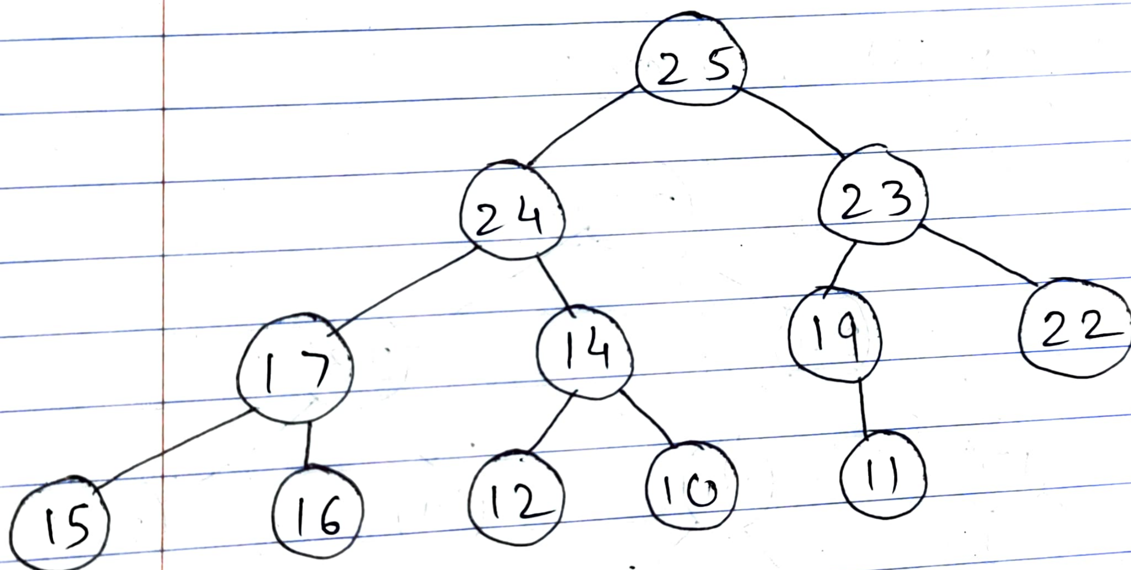
∴ After extract-max:



Array:

< 25, 24, 23, 17, 14, 19, 22, 15, 16, 12, 10 >

e) We insert 11 right after the last leaf node in the max-heap tree in 1(d), ~~aft~~ after running max-heapify on it, we get:



Array:

⟨25, 24, 23, 17, 14, 19, 22, 15, 16, 12, 10, 11⟩

3. $A = \langle 1, 5, 9, 6, 3, 2, 8, 7, 4, 0 \rangle$

a) Quick Sort [Partition method used → last element is pivot ($A[r]$))

i)

i									r
1	5	9	6	3	2	8	7	4	0
j									

↓ after partition(),

i								j	r
$\boxed{0}$	5	9	6	3	2	8	7	4	1

ii) Now, 0 is in the right position. We solve ~~from~~ for the ~~2nd~~ subarray from 2nd element to last because there is no subarray to the left of 0 (first element) (pivot's correct place)

∴ New array :

i j r
5 9 6 3 2 8 7 4 1

↓ after partition()

i j r
$\boxed{1}$ 9 6 3 2 8 7 4 5

Similar to (i), 1 is in its right place,
∴ We solve for right subarray

iii)
j r
9 6 3 2 8 7 4 5

↓ after partition

j r
3 2 4 $\boxed{5}$ 8 7 9 6
i

Now we have 2 subarrays →
<3, 2, 4> and < 8, 7, 9, 6 >

Solving for <3, 2, 4> first

iv) i j r j r
3 2 4 → 3 2 $\boxed{4}$
i

4 is in right place. We sort
for <3, 2>

j r i r
3 2 → $\boxed{2}$ 3
j

∴ & Since 3 is the only element left, we have its final position too.

Now, sorting < 8, 7, 9, 6 >

v) j r j r
8 7 9 6 → ~~8 7 9 6~~
i
↓

$\boxed{6}$ 7 9 8
i j r

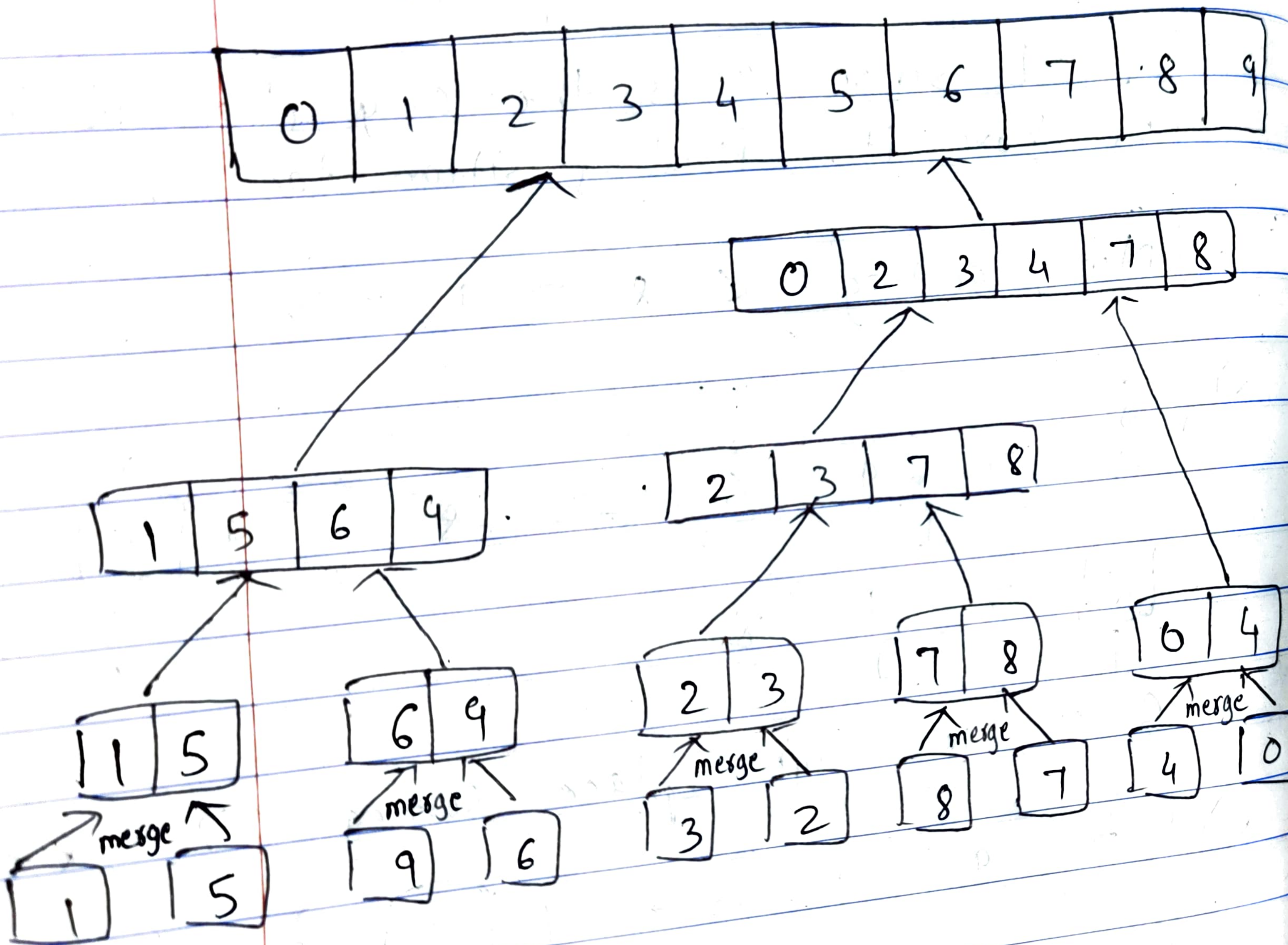
6 is in right place. Sorting <7, 9, 8>

vi) i j r j r
7 9 8 → 7 $\boxed{8}$ 9
i

Left & right subarrays are single element each.
∴ We have our sorted array from (i), (ii), (iii), (iv), (v) and (vi) as

< 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 >

b) Merge Sort



0 1 2 3 4 5 6 7 8 9
0 2 3 4 7 8
1 5 6 9
2 3 7 8
1 5
6 9
2 3
7 8
6 4
merge
merge
merge
merge
merge
1
5
9
6
3
2
8
7
4
0

Q3 c)

- Quick sort performs better for smaller datasets but the time complexity increases as the data increases while Merge sort performs the same for any type of data

- Quicksort's partitioning method can be modified depending on the nature of data to increase the efficiency. Thus if the nature of data is known, QuickSort's partitioning can be modified to perform better. On the other hand Merge sort will always divide the array into halves so the complexity always remains the same.

- In worst case scenarios, quick sort's partitioning can lead to unbalanced arrays -> which mean the pivot could always be the first or last element on the edge, thus making the time complexity **$O(n^2)$.** But Merge Sort will always be **O (nlog(n))**

- Quick sort does not require any additional space to perform the sorting therefore the algorithm is very space efficient while Merge sort requires O(n) to store the temporary array data.

Reference from CLRS 9.3,

```
{ SELECT (A, n, i)
      Divide A into 5 almost equal
          subarrays
      Find median of each subarray
          using insertion sort
      Recursively select median x of the
          medians found in the groups/
          subarrays
      Let L be smaller subarray, R be greater
          and x the median, the pivot
      if i =(index of x) then return x
      else
          if i<(index of x)
              SELECT (L, index of x, i)
          else
              SELECT (R, n-index of x, i-index(x))
}
```

This algorithm finds the k^{th} smallest element in an array.

∴ To find the median we write a driver function :

```
Partition (A, n)
    median = SELECT(A, n, ⌈n/2⌉)
    A0 = array of size ⌈n/2⌉
    A1 = array of size ⌊n/2⌋
    for i=0
        if A[i] <= median
            A0 = A0 + A[i]
        else
            A1 = A1 + A[i]
    return A0, A1
```

Runtime Analysis

For a particular x, out of total medians, there will be at least half medians greater than x and half smaller.

Also, for each of the $\lceil n/5 \rceil$ median, there will be 3 elements greater than x, except the last partition and the partition in which x is

∴ ~~We make $\frac{3n}{2}$ - 6 comparisons~~

∴ $\frac{3n}{10} - 6$ elements each are greater and smaller than x

∴ We call SELECT at most $\frac{7n}{10} + 6$ times.

∴ $T(n) = T(n/5) + T\left(\frac{7n}{10} + 6\right) + O(n)$

Let's assume that $T(n)$ works for linear time. cn

$$T(n) \leq c\lceil n/5 \rceil + c(7n/10 + 6) + an$$
$$\leq \frac{9cn}{10} + 7c + an$$

∴ $cn \leq cn + (-cn/10 + 7c + an)$

$\therefore$ We have the inequality

$$-\frac{cn}{10} + 7c + an \leq 0$$

$$\therefore \quad c \geq 10a\left(n/(n-70)\right)$$

Here, for any n such that $n > 70$, we can satisfy this equation.

$\therefore$ Our assumption is right

Worst case running time for SELECT is $O(n)$

The PARTITION for loop will take $\theta(n)$

$\therefore$ For the whole algorithm, we have

$$T(n) = O(n) + \theta(n)$$
$$= \cancel{\theta n}\ O(n)$$

Q4] Alternate answer using heaps

1. We maintain two heaps, one min-heap and one max-heap such that all the elements in the min-heap and always greater than all the elements in the max-heap
2. We traverse through an unsorted array or receive stream of numbers. We process each incoming number/element one by one
3. By default, add elements in the max-heap which maintains the smaller elements
4. When the difference in size of both the heaps becomes more than one, extract the max element from the max-heap and add it into the min-heap.
5. Also, when the two heaps arrive at a state where an element in the max-heap is greater than any element in the min-heap, extract the max-element from the max-heap and add it to the min-heap. The extracted element will always be the element which was violating **point number 1**
6. This way, at the end, we will have easy access to the middle two elements of the array/stream traversed so far. (Root element of the max-heap and root element from the min-heap) since we are maintaining the size of both the heaps as well as the property that all elements in max are lesser than all elements in right
7. Now the root elements can be used to find median depending on the size difference of the two heaps
8. If both heaps have the same size in the end, median is the average of both roots
9. If size is different, the median is the root of the heap with bigger size.